The Researcher's Guide to Code Organization

efrén cruz cortés

TA: Yangyang Li

Introduction

Does this sound familiar?

- I can’t find my script!
- My script worked last week but now it’s broken. I didn’t touch it.
- My script works perfectly in my computer but not when I share it with my colleague.
- I wrote this a year ago, and I can’t remember what it does.
- These parameters worked for me, feel free to change them in the script.

- What does this file do? I’ll just delete it…

Motivation

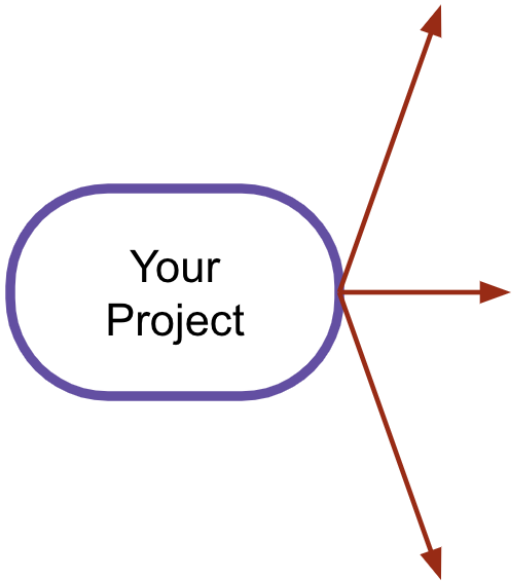
- A code-based research project is composed of many elements outside internal code logic, and yet, these are not taught in classes!
- We need a place to understand how all these elements come together, why they are necessary, and the principles behind them.
- Proper organization will facilitate collaboration, reproducibility, and other important elements of research.

A bit of effort now will save you sweat and tears in the future!!

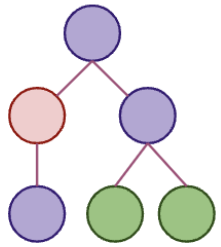
Overview

- Day 1 will focus on overall project organization principles and tools.
- Day 2 dives into organizing principles within a script.

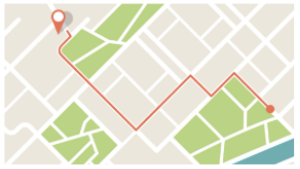
Day 1- Overview



Structure



Paths



Naming

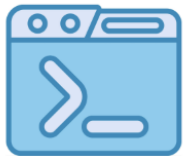
Documentation

File Types

[T] YAML

Version Control


The Command Line



IDEs


R Studio
Spyder
jupyter
Pluto.jl

Environments

CONDA
yarn


General Principles

- Organization
  - Separation of Concerns
  - Modularity
  - Hierarchy
  - Containment
- Efficiency
  - Usability
  - Portability
  - Automation
  - Scalability
- Collaboration
  - Transparency
  - Reproducibility
  - Version Control

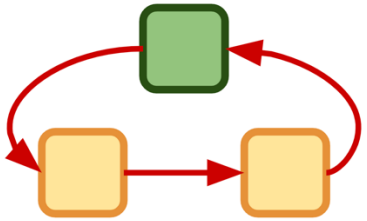
Project Structure

Identify your project structure

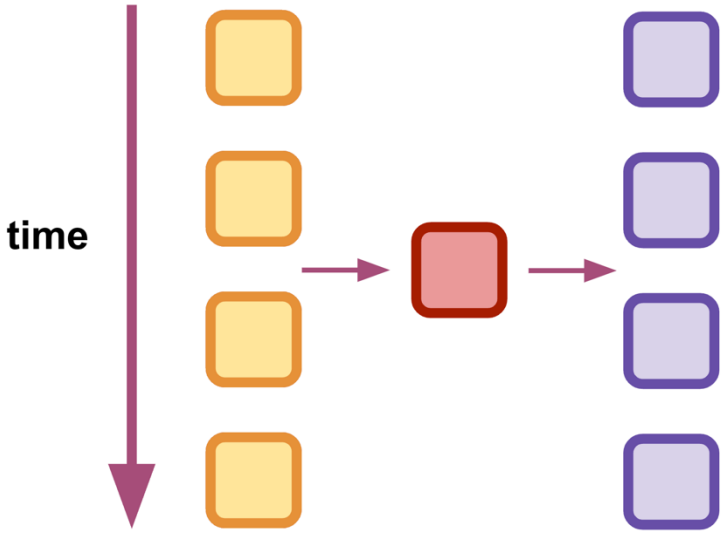
- Does your project use data?
- If so, at which stage?
  - At the beginning (training, preprocessing, …)
  - At the end (calibration, validation, …)
  - Throughout (sensors, inference, …)
- Is your project iterative in nature?
- Does your project branch into possible models / analyses?

A simple project taxonomy*

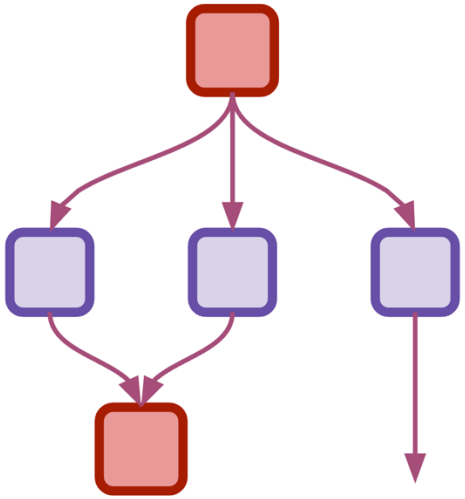
Linear



Iterative



Recurrent



Branching

*I couldn't find an official taxonomy / framework for research project structures, so these examples are limited to my own experience. -ecc

Examples

In reality, most projects are iterative in nature as we improve our own understanding and as we calibrate with data.

These are generic possibilities, but of course research is more complex:

- **Linear:** batch learning, descriptive statistics
- **Recurrent**: online learning, sensor data
- **Iterative**: physics simulation, agent-based models
- **Branching:** model selection, by-case analysis

Exercise

Grab a piece of paper and sketch the structure of your project.

- It’s OK if it isn’t perfect, if it doesn’t fit the shapes above, or it is a strange mix.

Directory Structure

Keep the project contained

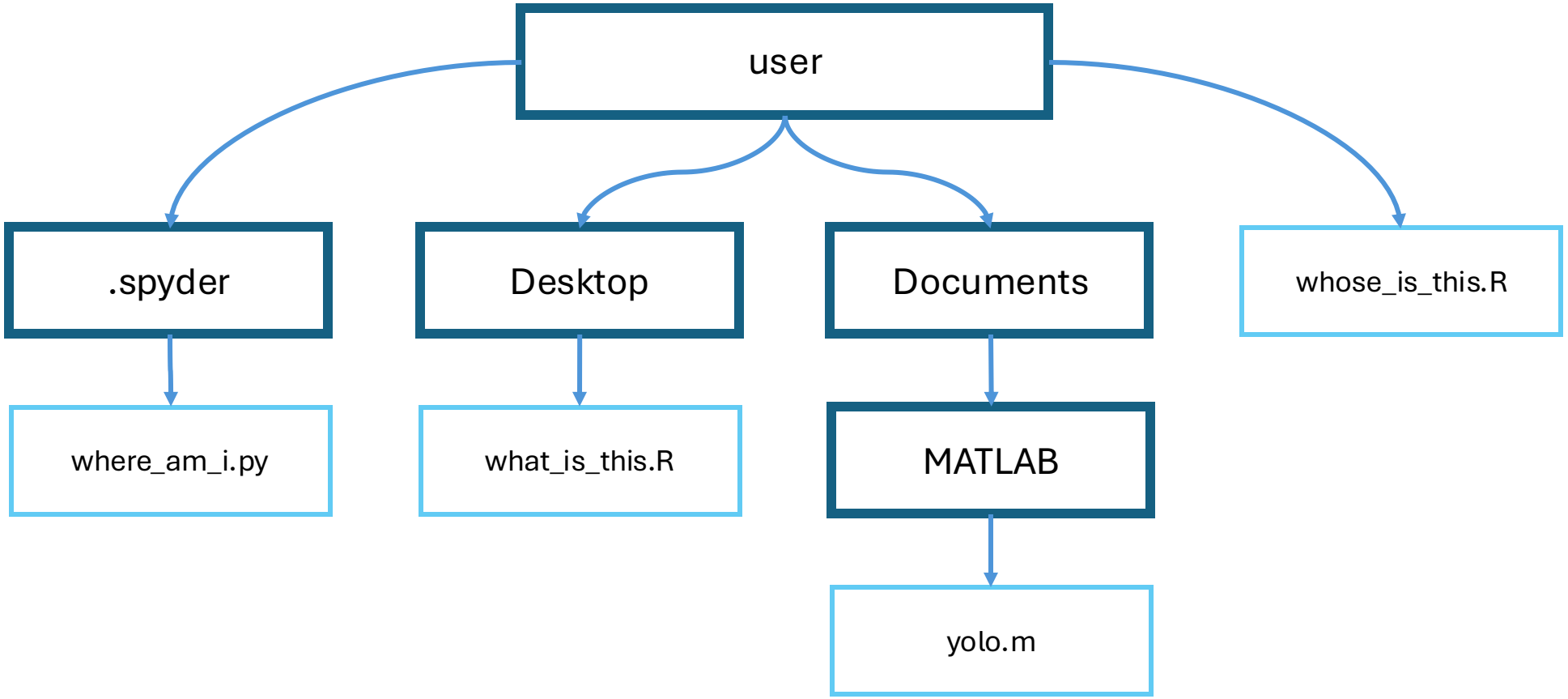
By default, all materials for your project should go inside one project directory.

(Imagine the chaos of saving all your .doc files in the Desktop)

Plan your project directory based on your project structure.

Bad!

See example folder ***Example-Bad***



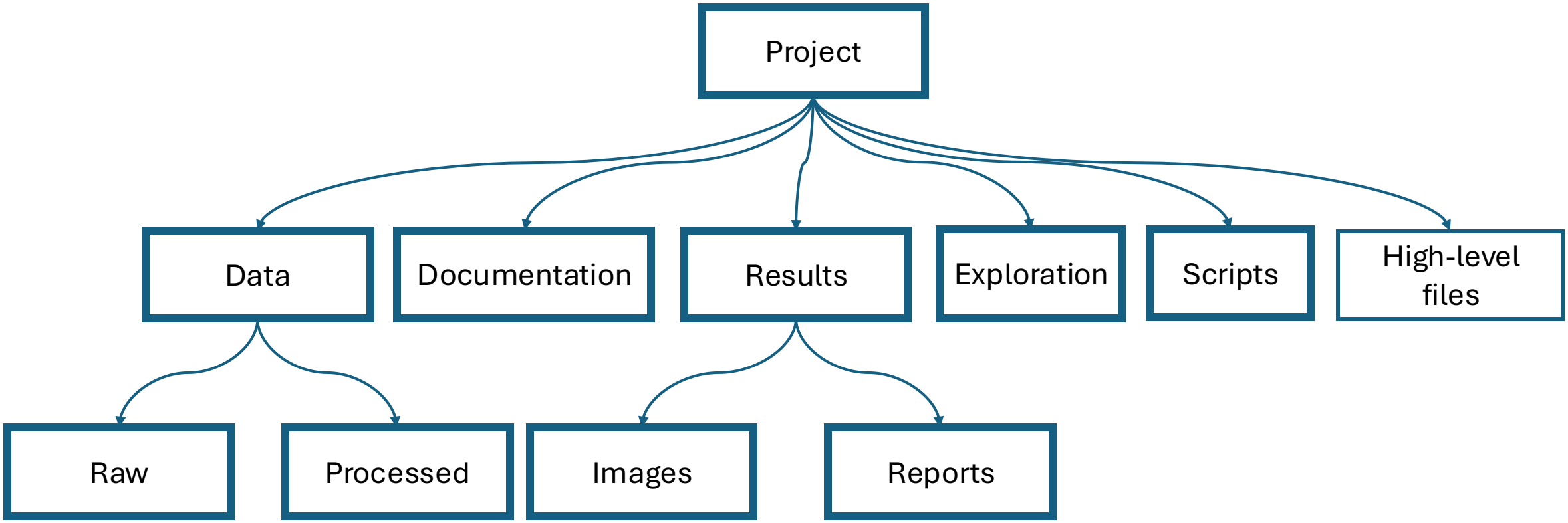
Separation of Concerns

Separation of concerns is the general principle in which pieces of a project should be separated according to their distinct form and function.

For example, data is separated from “code” files.

Within coding, preprocessing is separate from analysis, which is separate from visualization.

Good! Common organization paradigm



Example

See example folder ***Project_1***

No size fits all

The previous organizational paradigm is common and a good starting point, but your project may ask for a different structure.

General Principles to follow:

- Containment
- Hierarchy
- Separation of Concerns

Tree structures

Note that you won't always be able to visualize a tree as above. Sometimes the tree structure is written out using tabs and/or lines:

Project

| - Data

    | - Raw

    | - Processed

| - Scripts

| - Results

    | - Figures

    | - Reports

| - high_level_file

Exercise

Grab a piece of paper and create a draft of your directory structure based on the first exercise.

- Think about where you will save files by function and by type.
- You can draw a tree diagram or write it out in linear format as above.

Then, create the respective directories in your computer!

Note on literature

You may be tempted to add supporting literature (research papers, etc.) to your project directory. However, you will often use overlapping literature for different projects.

Instead, I recommend you organize literature elsewhere and use a reference management tool like **Zotero**.

General Principles

- Containment
- Separation of concerns
- Modularity
- Hierarchy
- Efficiency

Intermezzo 1 - Paths

The current working directory

- A directory is a location in your computer (a folder in the desktop, for example)
- When running a script, the *current working directory* (cwd) is the location from where the script is run (set by IDE or terminal)
- Not all scripts are at the same location as the cwd, but we should aim for them to be “close by” (see next slide).

Absolute vs relative paths

- The *absolute* path of a file is the “address” from the root directory of your computer.
  - /Users/efren/Desktop/Example/some_file.txt

- A *relative* path is the address as viewed from a different starting point.
  - If I’m on the Desktop already:
  - /Example/some_file.txt

Where’s your Desktop?

Note:

On some new laptops, your Desktop folder may be automatically in either iCloud or OneDrive.

- This means your files are not saved in your computer but in the cloud!
- Just make sure you know where everything is, and are intentionally using one option or the other.

Use relative paths!

- Common relative path indicators:
  - “.” indicates current directory
  - “..” indicates parent directory
  - “A/B” indicates A is a sub-directory of B
    - “\” for Windows
  - “/” at the begining means “root directory”
    - “C:” for Windows

- Assume I’m on the desktop:
  - ./Project_1/Data
  - ../Downloads

Path Standardization

Paths are written differently for different operating systems (forward / backward slash being the main culprit)

- Many languages have a tool to help you standardize paths.
- In python, for example, `pathlib` does that for you.

Example

See folder ***Example_Paths***

General Principles

- **Portability**
  - You can move your folder around and it won't break.

- **Collaboration**
  - You can share your project with other people!

Exercise

Assume you are situated in “Project_1”. Change the following absolute paths to relative:

- /Users/you/Desktop/Project_1/data/your_data.csv
- /Users/you/Desktop/Project_1/auxiliary_file.py
- /Users/you/Desktop/Project_2/
- /Users/you/Downloads

Naming

Conventions

- Be **specific** and be **descriptive**
- Be **consistent**
- Name according to specific function, stage, date, etc.
- Avoid generic names like *data.csv*, *function.py*, *main.py*.

Conventions – versioning

- Avoid file names with version suffixes like:
  - *my_file_final.R*
  - *my_file_final_final.R*
  - *my_file_final_final_FOR_REALZ_THIS_TIME.R*
- Try to avoid manual versioning (we'll talk version control soon)
  - *my_file_v1.m*
  - *my_file_v2.m*
- For data, dating is better (see next slide)

Conventions - dating

- Use the standard date format
  - YYYY-MM-DD
  - YYYY_MM_DD
- Most languages have a function to obtain current date/time.
- Get in the habit of appending date to data files names as either prefix or suffix.

Conventions – casing for legibility

- Recommended:
  - snake_case: *my_file.py*
  - camelCase: myFile.js
  - mixed: *heartData_Evanston_2025_02_24.csv*

- Acceptable:
  - hyphenated-case (usually for folders): *project-1*

- Avoid:
  - spaced names: *project report.pdf*
  - dotted*: *heat.data.evanston.csv*

* Some authors recommend dotted naming. I discourage it as it is quite confusing and unacceptable under some language conventions.

General Principles

- Usability
- Portability
- Automation
- Collaboration

Documentation

Where to document

- A README file for the whole project.
- Specific observations / thoughts, etc. (appropriately placed in directory tree structure).
- Inside your script.

README file

- Overall description of the project
- General requirements
- Other general information (structure, explanations, semantics, conventions)
- Example usage

Example

See example ***Project_2***

General Principles

- Reproducibility
- Collaboration

Intermezzo 2 – File Types

What should README be?

- You may have seen README files as:
  - *README.txt*
  - *README.md*
- The latter one is a markdown file (nicely formatted, used often in the web).

Different data types

- Do not be afraid of different file types, you will encounter and work with many, and it’s fine.
- Learn what each does and how it interacts with others in your project

Data: language specific vs agnostic

- You may be used to saving your data in language-specific formats:
  - .pickle, .pkl, .mat, .rds, .rdata, etc.
- These formats are fast and convenient, but are not portable and easily shareable.
- If you want to share data, aim for language-agnostic formats, at least for finalized data:
  - .csv, .json
- If you don’t know what a file is, look it up!

Auxiliary files

- You will encounter, and use, a variety of auxiliary files
  - To store directory information, configuration parameters, “secrets” (API keys, e.g.), requirements, version control info, etc.

- We’ll talk about these more in Day 2, but pay attention:
  - .env (usually for secrets and other environment variables)
  - .yaml, .toml (usually for configuration parameters, requirements)
  - .gitignore (which files to ignore by Git, more on this later)

Seeing hidden files

- Invisible files start with a “.”.
  - .gitignore
  - .env

- You can see them in your computer, while on your file explorer:
  - Mac: command + shift + .
  - Windows: File Explorer -> View -> Hidden items

Creating files

- You will need an editor to create and edit those files.
- IDEs help you with this. More on them later.

Exercise

Experiment with the IDE you are used to (Jupyter Lab, RStudio, etc.) and see what type of files you can create and edit.

Make a mock README.md file.

General Principles

- Portability
- Reproducibility
- Collaboration

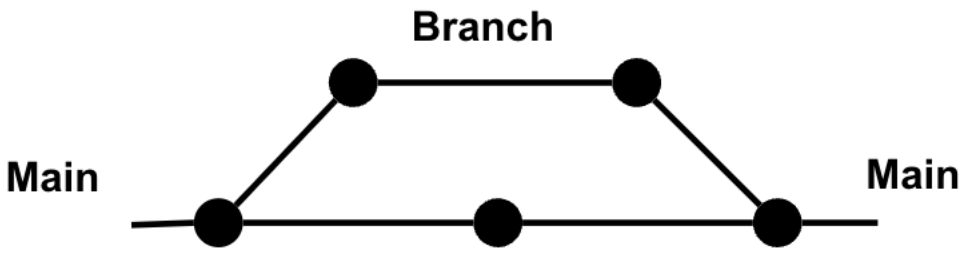
Version Control

What and why?

- Version control is any protocol used to keep track of versions of your code (or other materials) as you develop it.
- A version control system is any specific software that helps you achieve the above.
  - Helps you avoid manual version control: *file_v1.py*, *file_v2.py*, etc.
- The most common vcs is Git
- Using a version control tool is indispensable for efficient and robust collaboration!

Example

- Each point is a new version of your code. You can keep track of these on the central line.
- You may want to experiment with changing some aspects of your code, so you branch off, and later “merge” with the main line.
- Alternatively, you may share with a collaborator, who will make changes, and then merge.



Git

- We don’t have time to help you install and learn git, but we have materials here.
- It’s pretty easy to use, and to get started you need to learn less than 10 words:
  - init, status, add, commit –m, rm, rm --cached

- (Live demonstration)

GitHub

- GitHub is a website that allows you to:
  - Host repositories online
  - Perform Git actions through a graphical interface

- GitHub Desktop is a pretty handy app to keep your local and remote repos synchronized.

.gitignore

VERY IMPORTANT

The *.gitignore* file tells Git which files to ignore. This is a very important file if you use GitHub or another public repository to avoid sharing **secret information** (for example API keys) online.

You can find *.gitignore* templates for your favorite programming language here.

General Principles

- Collaboration
- Version Control
- Reproducibility

Intermezzo 3 – The Command Line

The command line interface

- Text based interface to communicate with your operating system
  - Terminal
  - Console
  - Prompt

- You will certainly need it for your projects, but don’t be afraid, you don’t need to learn everything about it.

Who’s afraid of the CLI?

- You just saw me use the terminal for my Git commands.
- Similarly, many tools will require you to run commands from the CLI, if anything just for installation.
- Usually, these tools have clear instructions on how to use it and what to run.
- You don’t need to learn everything now. Important steps:
  - Get comfortable with opening and closing it
  - Learn how to change directories

Integrated Development Environment

IDEs

Integrated Development Environments (IDEs) have many advantages besides editing scripts

- Enhance productivity
- Facilitate project management and Organization
- Help with version control
- Help with debugging

Examples

- Visual Studio Code
  - Most versatile
  - Supports many languages
  - Supports many file formats
  - Takes a bit of extra time to get used to
- Jupyter Lab
  - Ju-py-(te)r: supports Julia, Python, R
- Others:
  - RStudio – For R. Best for R and de-facto.
  - Spyder – Like RStudio for Python
  - Pluto – Interactive like Jupyter, but nicer.

Choose your own adventure – but choose well!

- If your language is proprietary (like MATLAB), you don't have a choice.
- If you are only working in R → RStudio
- New-ish to python:
  - Spyder if using *.py* scripts, want to explore variables, variable structure, easily peak into class methods and properties, etc.
  - Jupyter if using *.ipynb* notebooks, are doing exploration, or are preparing educational materials.
- More experienced, multi-language, and a bit of patience:
  - Visual Studio Code.
  - I'm a convert to VS Code. It is fast, helps you easily create auxiliary files of all types, supports many languages, etc.

Aside note on Jupyter notebooks

- Jupyter notebooks are popular because of their pedagogical value: it is easy to present step by step procedures.
- If you use them, use them for pedagogical reasons or for exploration and experimentation.
  - Make sure you add a folder akin to “exploration notebooks” to your directory tree!
- But, you should aim to have full *.py* scripts as your final result.
  - Containment.
  - Consistency
  - Reproducibility.
  - Automation.

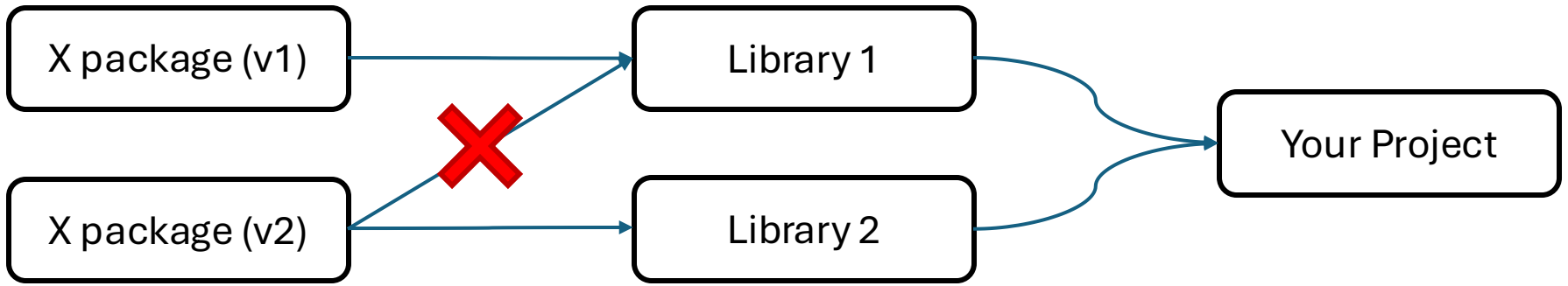
Environments

Environments

- An environment is an isolated space, generally self-contained, containing dependencies necessary to run a project.
- Dependencies include libraries, packages, and modules.
- Environments contain specific versions of the project's dependencies.
- They avoid conflict among different dependencies and their versions.

Dependency hell

- You can’t build everything from scratch. You will inevitably depend on libraries built by others.
- These are built mostly independently, so they can conflict with each other.
- When dependencies are in conflict with each other, your project may encounter unexpected errors and become impossible to run.



The proprietary paradise

If you are a MATLAB user (or use another proprietary language) you may not be used to this concept, since most packages are designed by the company and kept consistent to avoid these conflicts.

Environments and package managers

- Each language will have a tool to create environments, as well as to manage packages within an environment.
- These packages are often downloaded from a repository.
- Packages, repositories, and managers have varying degrees of security and security checks, so be careful when installing.

Environment cheat sheet*

Language	Terminology	Environment Creation Tool	Package Repository	Package Manager	Requirements Files
Python	• Virtual environments • Conda environments	• venv • conda	• PyPI • Conda	• pip • conda, mamba	• requirements.txt • environment.yaml
R	• Project environments	• renv • (.RProj)	• CRAN	• renv • install.packages()	• renv.lock
JavaScript	• node_modules	• npm • yarn	• npm registry	• npm • yarn	• node_modules • .pnp.cjs
Julia	• Project environments	• Pkg	• Julia general registry	• Pkg	• Project.toml • Manifest.toml
Rust	• Cargo projects	• cargo	• crates.io	• cargo	• Cargo.toml • Cargo.lock
MATLAB	• N/A	• Path management • Add-on manager	• MATLAB central file exchange	• Add-on Manager	• N/A

* I'd love to improve on this cheat sheet, if you have suggestions let me know!

Example

- Demonstration – setting up a conda environment
- Demonstration – setting up an evironment with renv

General principles

- Reproducibility
- Consistency
- Collaboration

Final Exercise

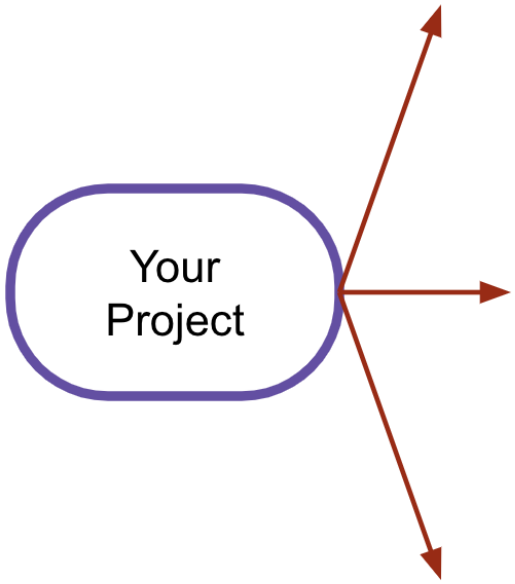
Chaos

Go to the Chaos folder, reorganize it and rename files based on the principles we learned today.

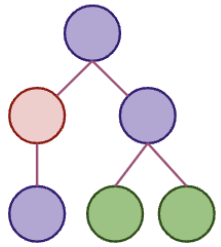
- All files in this example folder pertain to the same project.
- The mock files are empty, they are there just for practice.
- If there are folders or files that you think must be added, feel free to do so!

Recap

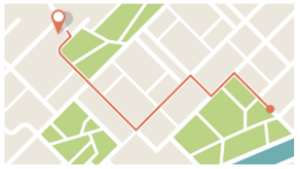
Day 1- Overview



Structure



Paths



Naming

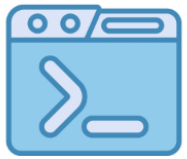
Documentation

File Types

[T] YAML

Version Control


The Command Line



IDEs


R Studio
Spyder
jupyter
Pluto.jl

Environments

CONDA
yarn


Questions?

Other resources

This cool post on file structure:

- https://mitcommlab.mit.edu/broad/commkit/file-structure/

This workshop by our very own data management team:

- Tobin_Magle_Data_Management_Slides

Related workshops by our data science team:

- The command line
- Conda environments
- Git and GitHub